In Proc. of *the 30th IEEE Conference on Decision and Control*,
Brighton, UK, Dec. 1991, pp. 1819–1823.

On the planning problem in sequential control

Inger Klein
Dept. of Electrical Engineering,
Linköping University
S-581 83 Linköping,
Sweden

Christer Bäckström
Dept. of Computer Science,
Linköping University
S-581 83 Linköping,
Sweden

June 2, 1994

Abstract

Sequential control is a common control problem in industry. Despite its importance fairly little theoretical research has been devoted to this problem. We study a subclass of sequential control problems, which we call the SAS-PUBS class, and present a planning algorithm for this class. The algorithm is developed using formalism from artificial intelligence (AI). For planning problems in the SAS-PUBS class the algorithm finds a plan from a given initial state to a desired final state if and only if any plan exists solving the stated planning problem. Furthermore the complexity of the given algorithm increases polynomially with the number of state variables.

1 Introduction

Planning is to sequential control what designing a controller is to regular control theory. Using a rather coarse model (typically finite state) the planning problem is always solvable in principle, but when the size of the problem increases it soon gives rise to complexity problems. Hence we propose to study subclasses where we retain feasibility. In this paper we study the SAS-PUBS class of planning problems which is defined below.

Sequential control is a common problem in industry. Almost all process plants contain a part which can be described as sequential control, for example when starting or shutting down a process plant. Despite its importance fairly little theoretical research has been devoted to this problem. Sequential control can be divided into two parts: planning and implementation. Planning is the problem of finding a plan, that is, a sequence of actions, which transforms a given initial state into a desired final state. A

plan can be illustrated by using the graphical notation GRAFCET, which has many similarities with Petri nets. A description can be found in the GRAFCET standard [22]. However, GRAFCET is a tool for presenting plans rather than developing plans although it can be a help in structuring the process. In the implementation phase the plan is implemented, i.e., the actual controller or sequencer is designed.

We will here consider the planning problem which so far mainly has been studied in artificial intelligence(AI). A considerable amount of work has been spent on developing planners, that is, programs which automatically generate plans. One situation where such a planner could be useful is after an emergency stop. The process can then be in a state which is not known on beforehand. We want the planner to develop a plan to take the system back into normal operation or into some safe state of the system. If the number of states is large it is not realistic to have precompiled plans for all possible initial states, and there is thus a need for automatic planning. The work on planners started with the General Problem Solver (GPS) [16] and the perhaps most well-known planner is STRIPS [8]. In STRIPS each operator, that is, action, is defined by a set of *preconditions*, an *add list* and a *delete list* of clauses. In spite of all the work that has been done in this area, most of the methods used today are heuristic. The problem with most planners is that usually nothing is known about their correctness and completeness, or about their complexity. Chapman [6] has developed a planner called TWEAK. He proves that TWEAK is correct, i.e., the resulting plan will in fact solve the stated planning problem, and complete, i.e., it will always find a plan if there exists any plan solving the problem. However, TWEAK does not always terminate. There are also a lot of papers on temporal logics [1, 20]. The majority of these papers discuss representation problems and do not analyze complexity or computability. This should, however, be important when designing a planner.

Sequential control can also be viewed as a subfield of *discrete event dynamical systems (DEDS)*. In contrast to the well-known models for dynamical systems, which can be described by differential or difference equations, there is not yet any unifying theory for DEDS. A considerable amount of work has been done in different areas to describe and analyze DEDS, and to develop controllers for DEDS. Models for DEDS have been developed based on temporal logic [21], queueing theory [10], and minimax algebra [7]. A DEDS is easily described in automata theory, and work in this area has been done by Ramadge and Wonham [18], Inan and Variaya [12], and others. Another way of describing DEDS is using Communicating Sequential Processes developed by Hoare [11]. Benveniste and others [5] has developed a language which can be used to simulate DEDS. Petri nets [17] have also been used to describe and analyze DEDS.

Different models are developed for different purposes, and it is probably not possible to find a method which works well for all problems, so we will most likely have to develop different methods for different classes of DEDS. Here, we will study a special case of sequential control where all actions are controlled by the agent and we have considerable knowledge about the actions.

Our formalism, *Simplified Action Structures* is based on work by Sandewall and Rönnqvist [19] where an action is described by its *pre-*, *post-*, and *prevail-conditions*.

These conditions correspond to the *delete list*, *add list*, and *preconditions* used in STRIPS [8]. We will focus on the actions instead of the states and use the mentioned conditions to find a plan without having to explicitly construct the state graph. This is an advantage since the number of states in the state graph is normally exponentially larger than the number of available action types.

For a class of sequential planning problems, the *SAS-PUBS* class to be defined below, we present a planning algorithm. A planning problem is in the SAS-PUBS class if it can be formulated using simplified action structures, where the state variable domains are *binary* and the set of action types (a sort of generic action) is *unary*, *post-unique* and *single-valued*. Unary means that every action affects only one state variable and post unique means that there are no alternative actions to use to achieve a given goal. The only way that minimal plans differ from each other is in the order in which the actions are performed. If the set of action types is single-valued, then no action must be performed more than once. For planning problems in the SAS-PUBS class the defined algorithm will always find a plan if and only if there exists any plan solving the stated planning problem. Furthermore, the found plan is correct, i.e., when executed, it will in fact transform the initial state into the final state and the complexity of the algorithm increases polynomially with the number of state variables. A typical action in this class could be to close or open a valve, or to switch on or off some device, for example a motor.

The SAS-PUBS class of planning problem might be considered as rather small, and of course our aim is to extend this class while retaining tractability. It is important to study the planning problem analytically and to study a subclass of sequential control problems where this is possible. We can then use our conclusions to develop heuristic planning algorithms for more general problems. It is also interesting to characterize 'simple' planning problems to gain insight into planning and develop an understanding of how the structure of a planning problem affects the solution.

In this paper we will describe the main ideas behind the above mentioned algorithm. For formal definitions and proofs the reader is referred to [13]. The organization of the paper is as follows: in Section 2 we present the formalism we will use and introduce the notions of actions and plans. In Section 3 the restrictions forming the SAS-PUBS class of planning problems are stated and Section 4 gives an intuitive description of the planning algorithm. Section 5 presents a simple example and Section 6 contains the conclusion.

2 A formalism for describing the planning problem

The world can be described by a state and a set of actions which transforms the state of the world into a new state. An action is usually performed by an agent; it has a duration in time and it has a result, i.e., it affects the state of the world in some way.

An agent can be, for example, a robot or a human being. The formalism, *simplified action structures*, presented here is based on work by Sandewall and Rönnquist [19], but somewhat simplified. The main advantage of using action structures instead of, for example, a finite state automaton is that the number of states is exponential in the number of state variables. It is also intuitively attractive to describe the actions and how they affect the state of the world.

We will use some concepts about relations, and the reader who is not familiar with relations and partial orders is referred to [9]. In [3, 4, 13] the formalism is described in more detail.

2.1 States

The state of the world is described by a state vector x of dimension n, i.e., $x = (x_1, \ldots, x_n)$. Each state variable belongs to a discrete, finite set $\mathcal{S}_i$ and thus $x \in \mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n$. In this paper we will only consider binary state variables, but $\mathcal{S}_i$ may of course be any finite set. Each state variable set is extended with the value *undefined* (u) and *contradictory* (k). The undefined value can be interpreted as 'don't care'. The contradictory value is added for technical reasons only. A state x is called a *partial state* if the undefined or contradictory value is allowed for its components.

2.2 Action types and actions

Examples of actions could be $move\_workpiece$, where a robot moves a workpiece from a work-station to storage, $read\_input\_channel$, where a computer reads an input channel, and $drive\_to\_shop$, where a human being drives to a shop.

An action is formally described by two concepts, an *action label* and an *action type*. The action type can be interpreted as a generic action, and the action label is put on an action type to distinguish between different actions of the same type. An action can only occur once, and is a particular instantiation of an action type. The same action type can occur several time with different labels.

An action type a is defined by its *pre-*, *post-* and *prevail-condition*. The pre-condition $b(a)$ states what must hold when the action starts, the post-condition $e(a)$ what holds when the action ends, and the prevail-condition $f(a)$ what must be true during the performance of the action but which is not affected by the action. These conditions are partial states which do not contain the contradictory value for any state variable.

Consider, for example, the action type *refuel* where an aircraft is refueled. Here, the pre-condition is that the tank of the aircraft is empty, and the post-condition is that it is full. Thus the pre- and post-conditions describe what is changed by the action. When refueling the aircraft it is important that the aircraft is grounded, i.e., an electrical connection between the aircraft and the ground is established to eliminate the voltage difference. Thus, there is a condition which must be fulfilled when the action

is performed, but which is not affected by the action. This is the prevail-condition and for the action type *refuel* the prevail-condition is that the aircraft is grounded.

2.3 Planning

A plan $\langle \Psi, \rho \rangle$ from x^o to $x^\star$ is a set of actions Ψ and a partial order ρ on the set Ψ. The order ρ is the execution order which tells in which order the actions in the set Ψ should be performed in order to transform the initial state x^o into the final state $x^\star$.

The *planning problem* can now be stated as follows.

- Given a set of action types $\mathcal{H}$, a state space $\mathcal{S}$, an initial state $x^o \in \mathcal{S}$ and a final state $x^\star \in \mathcal{S}$, find a plan $\langle \Psi, \rho \rangle$ from x^o to $x^\star$.

We say that a plan is *minimal* if there is no plan from the given initial state to the desired final state containing fewer actions. The persistence handling is the same as the STRIPS assumption [8], namely that nothing changes unless explicitly stated in the pre- and post-condition.

3 The SAS-PUBS class of planning problems

In this section we describe the restrictions forming the class of planning problems we focus on. The formal definitions can be found in [3, 4, 13].

The class of planning problems defined so far is called the *SAS class*, where *SAS* stands for Simplified Action Structures. To form the class of planning problems we focus on in this paper we introduce some further restrictions:

- all state variables are *binary*
- each action affects only one state variable (*unary*)
- no two different action types can change a particular state variable to the same value (*post-unique*)
- no two different action types have different but defined (not the undefined value) prevail-conditions for the same state variable (*single-valued*)

This class of planning problems is called the *SAS-PUBS* (Post unique, Unary, Binary, Single-valued) class.

An example of a problem in the SAS-PUBS class is a process plant where some fluid is transported in pipes. In such a plant the typical action types would be to open or to close a specific valve. However, the restriction to single-valued sets of action types means that if there is one action type whose prevail-condition is that a specific valve is open, then there can be no action type whose previal-condition is that this valve is closed.

4 Planning for planning problems in the SAS-PUBS class

Finding a plan from an initial state x^o to a final state $x^\star$ is equivalent to finding a path in the state graph from x^o to $x^\star$. One way of finding a plan is of course to search the state graph. Such a planner would always succeed in finding a plan if there is any plan at all. However, the complexity of such a planner will increase exponentially with the number of state variables, and is thus not suitable when the number of state variables is large. Our approach is instead to develop more specialized planners tailored for different classes of planning problems. For a full presentation of complexity theory the reader is referred to, for example, Mendelson [15].

Our main idea is to use the structure of the SAS-PUBS class to find and order the actions without having to explicitly construct the state graph. Instead we use the pre-, post- and prevail-conditions to find which actions to perform and in what order these actions should be performed. Here we will only describe the process intuitively. The formal definitions can be found in [3, 4, 13].

For planning problems in the SAS-PUBS class, planning can be divided into two parts: finding the *set of necessary and sufficient actions* and finding the execution order *precedes*. The set of necessary and sufficient actions (Δ) consists of two sets: the set of *primary necessary* actions and the set of *secondary necessary* actions. The process of finding these two sets can be described as follows:

- The *set of primary necessary actions* P_0 is found by checking the difference between the initial state x^o and the final state $x^\star$, and search for the actions whose pre- and post-conditions corresponds to this difference. These actions form the set of primary necessary actions.

- The *set of secondary necessary actions* is found by an iterative procedure. If there where no prevail-conditions, i.e., any action could be performed in any state where the pre-condition of the action is satisfied, the set of primary necessary actions P_0 would contains all the actions needed to transform x^o into $x^\star$. Because of the prevail-conditions we can usually not perform these actions without performing some other actions to set their prevail-conditions. Suppose $a \in P_0$ such that the prevail-condition for the i^{th} state variable is not fulfilled in the initial state and that $x_i^o = x_i^\star$. Then we have to perform one action to temporarily change x_i in order to fulfill $f(a)$ and another action to change x_i back again to its required value $x_i^\star$. These two actions belong to the set of secondary necessary actions. Thus the set of secondary necessary actions contains set/reset pairs for some state variables x_i to assure that the i^{th} state variable is temporarily changed if required by the prevail-condition of some action a in the set of primary necessary actions. Furhtermore we can probably not perform these new actions without setting and resetting their prevail-conditions. This is an iterative procedure which will eventually stop because all needed set and reset actions are already included

in the set.

The set of necessary and sufficient actions Δ is now the union of the two sets above.

The next step is to find the execution order *precedes* (δ) which is a relation defined on the set of necessary and sufficient actions Δ. The relation precedes is constructed from two relations *enables* and *disables*.

- If a_1 'enables' a_2 then a_1 provides some part of the prevail-condition for a_2.
- If a_2 'disables' a_1 then a_2 destroys some part of the prevail-condition for a_1.

In both these cases a_1 should be performed before a_2. Putting these two relations together and taking the transitive closure gives the execution order δ.

Now $\langle \Delta, \delta \rangle$ is a plan from the initial state x^o to the final state $x^\star$ if the set Δ exists and δ is a partial order. If δ is not a partial order or if the set Δ does not exist then there is no plan transforming x^o into $x^\star$ using the available actions. Furthermore the found plan is minimal. This is stated in Theorem 4.1.

Theorem 4.1 For planning problems in the SAS-PUBS class $\langle \Delta, \delta \rangle$ is a minimal plan from x^o to $x^\star$ if and only if any plan from x^o to $x^\star$ exists using the available actions.

Proof: The proof can be found in [13]. □

Theorem 4.1 is proved by using concepts from discrete mathematics about relations and partial orders. Note that when writing $\langle \Delta, \delta \rangle$ it is implicit that δ is a partial order. The unordered actions in Δ may be performed in parallel.

In Appendix A we give an algorithm for finding a plan $\langle \Delta, \delta \rangle$ following the process described above. In [13] this algorithm is proven to be correct and complete, i.e., the algorithm will in fact find $\langle \Delta, \delta \rangle$ if any plan exists solving the stated planning problem, and otherwise it will fail. The complexity of this algorithm is proven to increase polynomially with the number of state variables. This should be compared with the number of states in the state graph which is exponential in the number of state variables. If the available actions is to close or to open a specific valve than each state variable corresponds to one valve, and the number of state variables is the number of valves in the plant.

Theorem 4.2 An algorithm finding $\langle \Delta, \delta \rangle$ if and only if there is any plan exists and the complexity of this algorithm increases polynomially with the number of state variables.

Proof: The proof can be found in [13]. □

We have here only considered planning problems in the SAS-PUBS class. In [14] an algorithm for planning problems in the SAS-PUS class can be found. In the SAS-PUS class we allow non-binary state variables. This algorithm is proven to be correct and complete, and the complexity is proven to increase polynomially with the number of available actions.

5 Example

In this section we apply our planning algorithm to a simple example. The problem is to refuel an aircraft using a mobile refuel vehicle. We define four state variables such that for any state $x \in \mathcal{S}$, the state x is interpreted as:

$$
\begin{aligned}
x_1 &= \begin{cases} 0 \text{ if the tank of the aircraft is empty} \\ 1 \text{ if the tank of the aircraft is full} \end{cases} \\
x_2 &= \begin{cases} 0 \text{ if the refuel vehicle is not at the aircraft} \\ 1 \text{ if the refuel vehicle is at the aircraft} \end{cases} \\
x_3 &= \begin{cases} 0 \text{ if the aircraft is not grounded} \\ 1 \text{ if the aircraft is grounded} \end{cases} \\
x_4 &= \begin{cases} 0 \text{ if the tank of the aircraft is open} \\ 1 \text{ if the tank of the aircraft is not open} \end{cases}
\end{aligned}
$$

When refueling the aircraft, it is important to eliminate the voltage difference between the aircraft and the refuel vehicle in order to avoid sparks. That the aircraft is grounded thus means that such an electrical connection is established, so grounding has nothing to do with whether the aircraft is airborne or not. There are seven action types in $\mathcal{H}$, and these are defined together with their pre-, post-, and prevail-conditions in table 1.

action type (h)	$b(h)$	$e(h)$	$f(h)$
refuel	$(0,u,u,u)$	$(1,u,u,u)$	$(u,1,1,0)$
move_vehicle_to_aircraft	$(u,0,u,u)$	$(u,1,u,u)$	(u,u,u,u)
move_vehicle_from_aircraft	$(u,1,u,u)$	$(u,0,u,u)$	(u,u,u,u)
ground	$(u,u,0,u)$	$(u,u,1,u)$	$(u,1,u,u)$
unground	$(u,u,1,u)$	$(u,u,0,u)$	$(u,1,u,u)$
close_aircraft_tank	$(u,u,u,0)$	$(u,u,u,1)$	$(u,1,u,u)$
open_aircraft_tank	$(u,u,u,1)$	$(u,u,u,0)$	$(u,1,u,u)$

Table 1: Definition of the action types for the example.

The initial state is $x^o = (0,0,0,1)$ and the final state is $x^\star = (1,0,0,1)$, i.e. we want to refuel the aircraft. The problem of finding a plan from x^o to $x^\star$ is clearly in the SAS-PUBS class. The set of necessary and sufficient actions is

$$
\begin{aligned}
\Delta(x^o, x^\star) = \{&\langle l_1, \mathit{move\_vehicle\_to\_aircraft}\rangle, \\
&\langle l_2, \mathit{move\_vehicle\_from\_aircraft}\rangle, \langle l_3, \mathit{ground}\rangle, \\
&\langle l_4, \mathit{unground}\rangle, \langle l_5, \mathit{refuel}\rangle, \langle l_6, \mathit{open\_aircraft\_tank}\rangle, \\
&\langle l_7, \mathit{close\_aircraft\_tank}\rangle\}
\end{aligned}
$$

where $l_1, l_2, \ldots, l_7$ are distinct labels from the set $\mathcal{L}$. The relation 'precedes' (δ) is given in figure 1, where a_k denotes the action with label l_k.

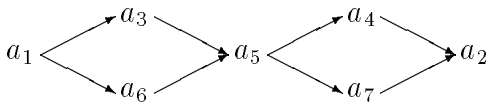

Figure 1: The relation 'precedes' (δ) on the set Δ.

6 Conclusion

We have solved the planning problem in the SAS-PUBS class by presenting an algorithm which is correct and complete, and complexity increases polynomially with the number of available actions. Even if the SAS-PUBS class can be said to contain only simple planning problems, it is important to develop theoretical results and thereby characterize what makes a planning problem simple. Furthermore we can use the knowledge gained for these simple problems as a source of inspiration when developing algorithms for more difficult problems.

A Algorithm

Here we present an algorithm for finding a plan $\langle \Delta, \delta \rangle$. First we define some functions and procedures used in the algorithm.

Definition A.1 We assume that the following functions and procedures are available:

Insert(a,A) Inserts the action a into the set A.

Find(A,i,x) Searches the set A for an action a such that $b(a)_i = x$. Returns a if found, otherwise returns **nil**.

Rfind(A,i,x) Like Find, but also removes a from A if it is found.

Warshall(M) M is a Boolean matrix representing a relation ρ. Returns a Boolean matrix representing the transitive closure of ρ. Uses Warshalls algorithm described in, for example, [2].

Algorithm A.1 [Plan]

Input: A, a set containing one action for each action type in $\mathcal{H}$, and x^o and $x^\star$, the initial and final states respectively.

Output: Δ a set of actions, and δ a partial order on Δ.

1 **Procedure** $Plan(A : set\ of\ actions; x^o, x^\star : state)$;

```
2   var
3     i :state variable index;
4     a, a', a_1, a_2 :action;
5     P, P', Δ :set of actions;
6     r :Boolean matrix;
7
8   begin
9     Δ := ∅;
10    P := ∅;
11
12    for i = 1 to n do
13      if x_i^o ≠ x_i^* then
14        a :=Rfind(A, i, x_i^o);
15        if a ≠ nil then Insert(a, P);Insert(a, Δ)
16        else fail
17        end {if}
18      end{if}
19    end{for};
20
21    while P ≠ ∅ do
22      P' := ∅;
23      for a ∈ P do
24        for i = 1 to n do
25          if f(a)_i ⋤ x_i^o then
26            a' :=Find(Δ, i, x_i^o);
27            if a' = nil then
28              a_1 :=Rfind(A, i, x_i^o);
29              a_2 :=Rfind(A, i, f(a)_i);
30              if a_1 = nil or a_2 = nil or e(a_1)_i ≠ f(a)_i or
31                 e(a_2)_i ≠ x_i^* then fail
32              else Insert(a_1, P');Insert(a_2, P');
33                   Insert(a_1, Δ);Insert(a_2, Δ)
34              end{if}
35            end{if}
36          end{if}
37        end{for}
38      end{for};
39      P := P'
40    end{while };
41
42    r := '|Δ| × |Δ| zero matrix';
43
44    for a ∈ Δ do
```

```
45        for a' ∈ Δ do
46          for i ∈ M do
47            if e(a)_i = f(a')_i then r(a, a') := 1 end;
48            if b(a')_i = f(a)_i then r(a, a') := 1 end
49          end{for}
50        end{for}
51      end{for};
52
53      δ := Warshall(r)
54      return ⟨Δ, δ⟩
55    end {Plan}
```

References

[1] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[2] S. Baase. *Computer Algorithms:Introduction and Analysis*. Addison Wesley, Reading, Massachusetts, 1988.

[3] C. Bäckström and I. Klein. Planning in polynomial time. In G Gottlob and W Nejdl, editors, *Expert Systems in Engineering: Principles and Applications. International Workshop.*, pages 103–118, Vienna, Austria, September 1990. Springer. Published as volume 462 of Lecture Notes in Artificial Inteligence.

[4] C. Bäckström and I. Klein. Planning in polynomial time. In Mary L Emrich et al., editors, *Methodologies for Intelligent systems: Selected Papers*, pages 125–129, Knoxville, Tennessee, Oct 1990. International Center for the Application of Information Technology. Paper presented at the ISMIS'90 conference.

[5] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Transactions on Automatic Control*, 35:535–546, 1990.

[6] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[7] G. Cohen, D.Dubois, J. P. Quadrat, and M. Viot. A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing. *IEEE Transactions on Automatic Control*, AC-30(3):210–220, 1985.

[8] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[9] A. Gill. *Applied Algebra for the Computer Sciences*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[10] Y. C. Ho and C. Cassandras. A new approach to the analysis of discrete event systems. *Automatica*, 19(2):189–208, 1983.

[11] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood CLiffs, New Jersey, 1985.

[12] K. Inan and P. Variaya. Finitely recurdive process models for discrete event systems. *IEEE Transactions on Automatic Control*, 33(7):626–238, 1988.

[13] I. Klein. Planning for a class of sequential control problems. Licentiate thesis 234, Department of Electric Engineering, Linköping, May 1990.

[14] I. Klein and C. Bäckström. Planning in polynomial time: The SAS-PUS class. Technical report, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 1991.

[15] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks, Monterey, California, 1987.

[16] A. Newell, J. C. Shaw, and H. A. Simon. Report of a general problem-solving program for a computer. In *Proceedings of an International Conference on Information Processing, UNESCO, Paris, France*, pages 256–264, 1960.

[17] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood CLiffs, N. J., 1981.

[18] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[19] Erik Sandewall and Ralph Rönnquist. A representation of action structures. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 89–97, Philadelphia, Pennsylvania, August 1986. Morgan Kaufman.

[20] Yoav Shoham. Temporal logics in AI: Semantical and ontological considerations. *Artificial Intelligence*, 33:89–104, 1987.

[21] J. G. Thistle and W. M. Wonham. Control problems in a temporal logic framework. *Int. J. Control*, 44(4):943–976, 1986.

[22] Union Technique de l'Electricite. *'GRAFCET' Functional Diagrams for the Description of Logical Control Systems*, 1982. French Standard NFC03-190.